Speeding up STM

Tim Harris, Mark Plesko,
Avi Shinnar, David Tarditi

15 July, 2005

Microsoft Research Cambridge

Introduction

- This work is about implementing atomic blocks

```
atomic {
  Object v1 = ht1.Remove(k1);
  ht2.Insert(k1, v1);
}
```

- The compiler & runtime system ensure that code inside the block runs atomically wrt other threads

Problem: isn't this all really slow?

Typically implemented by:

- Cloning all of the code that may run inside an atomic block
- Interposing on all of the instructions that may access the heap
- Log updates in a thread-private log
- Ensure that reads look into this log
- Commit the logged updates to the heap at the end of the atomic block

Research challenges

We'd like to avoid:

- Thread-private update logs
- Searching or sorting logs during commit
- Indirection headers on transactable objects
- Hash-based contention detection
- Allocation during [short] transactions

Research challenges

and at a higher level:

- Avoid logging altogether (no conflicts are possible on thread-local data)
- Avoid redundant logging (e.g. on multiple updates to the same data)
- Expose logging operations to compiler optimization (e.g. coalesce checks for log space)

Research challenges

Research challenges

Research challenges

Research challenges

Research challenges

Overview

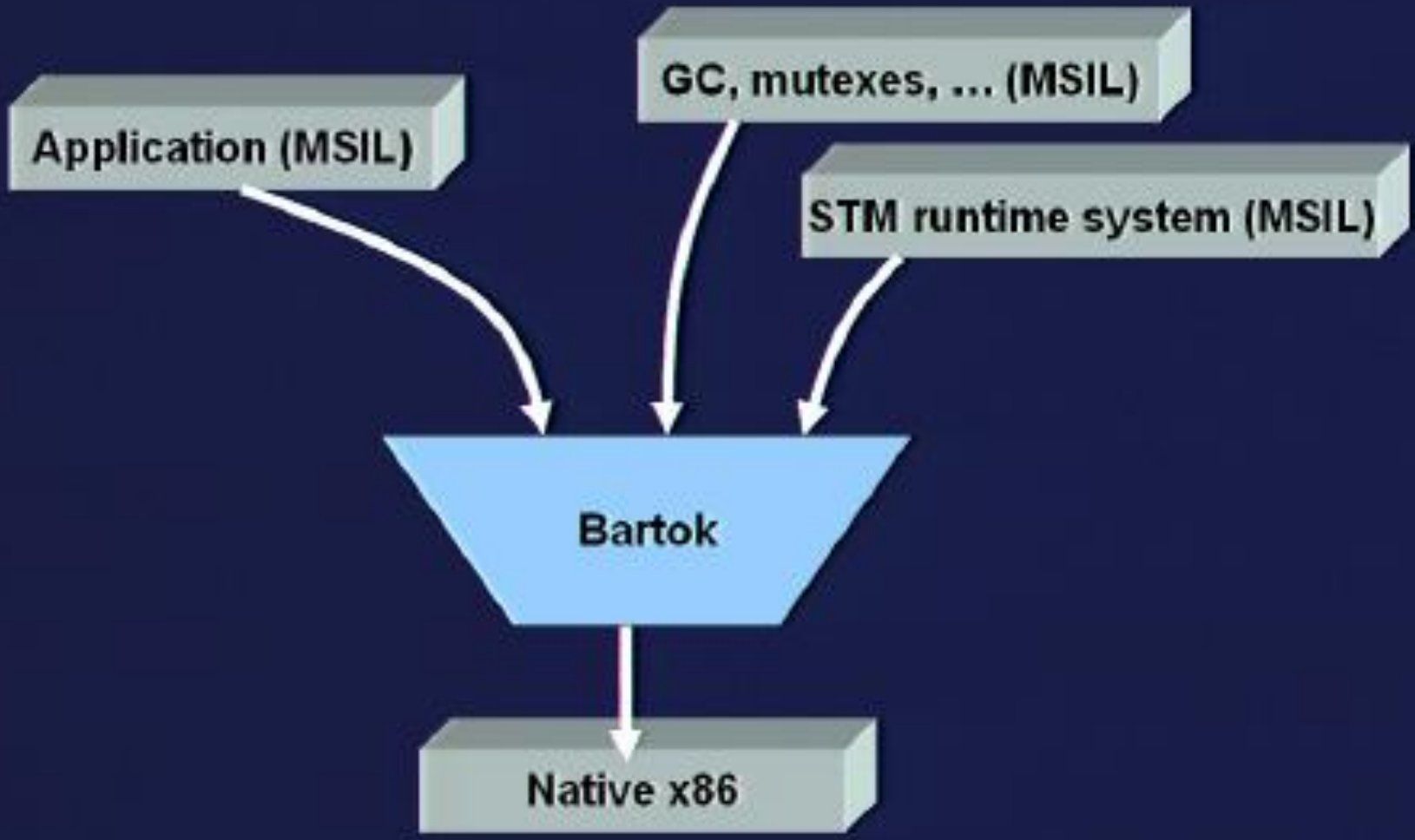
Introduction

Compilation

Runtime system

Initial results

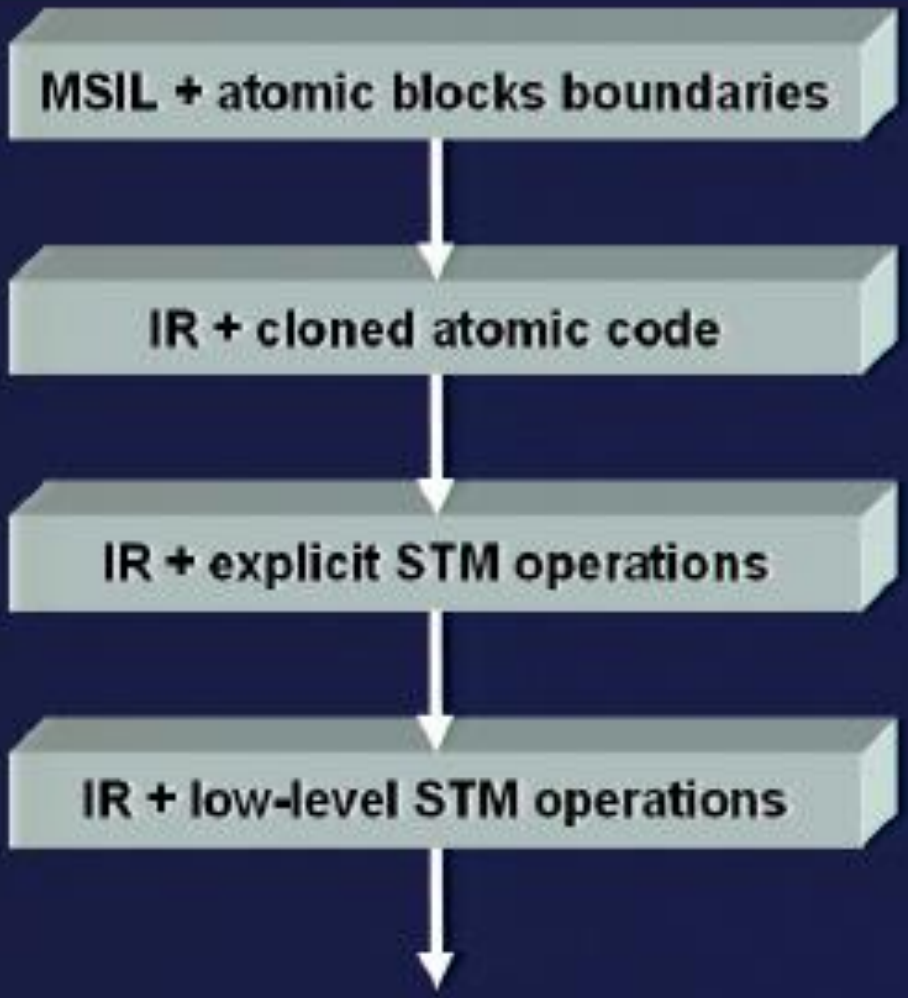
System structure



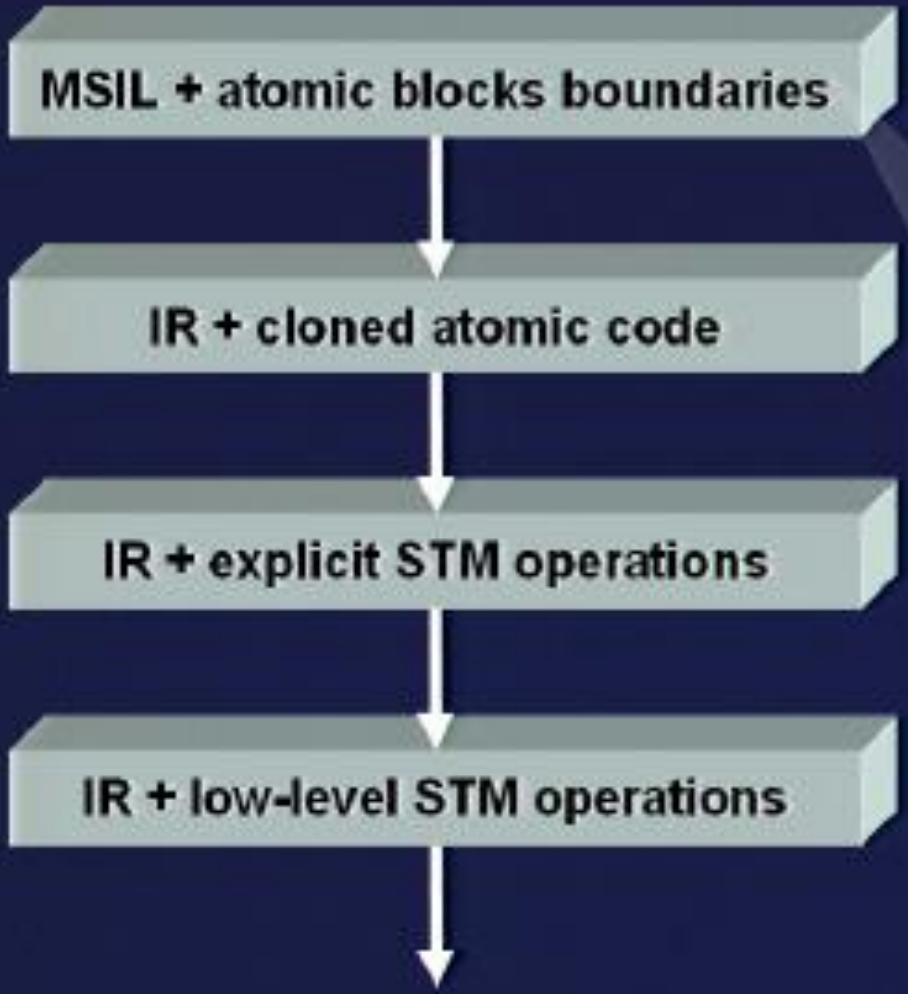
System structure

- The user writes code using atomic blocks
- These are translated into calls on the STM runtime
- The STM runtime is designed to avoid preventing optimizations within atomic blocks:
  - Updates are made directly to objects (=> no need to search a log on reads, no probs with sub-word data etc)
  - The compiler must insert calls on the runtime to:
    - Log the contents of locations that are overwritten inside an atomic block (=> allows roll-back)
    - Enlist locations that are read/updated in an atomic block (=> enables contention detection)

Compilation

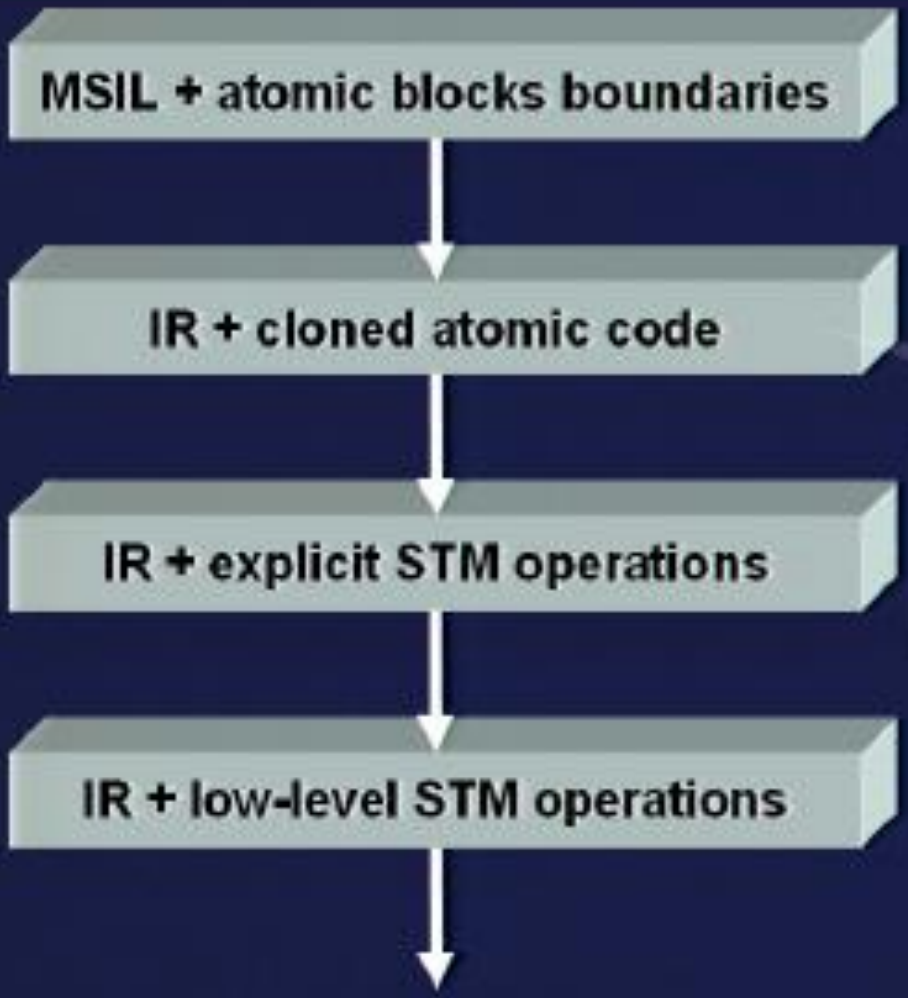


Compilation



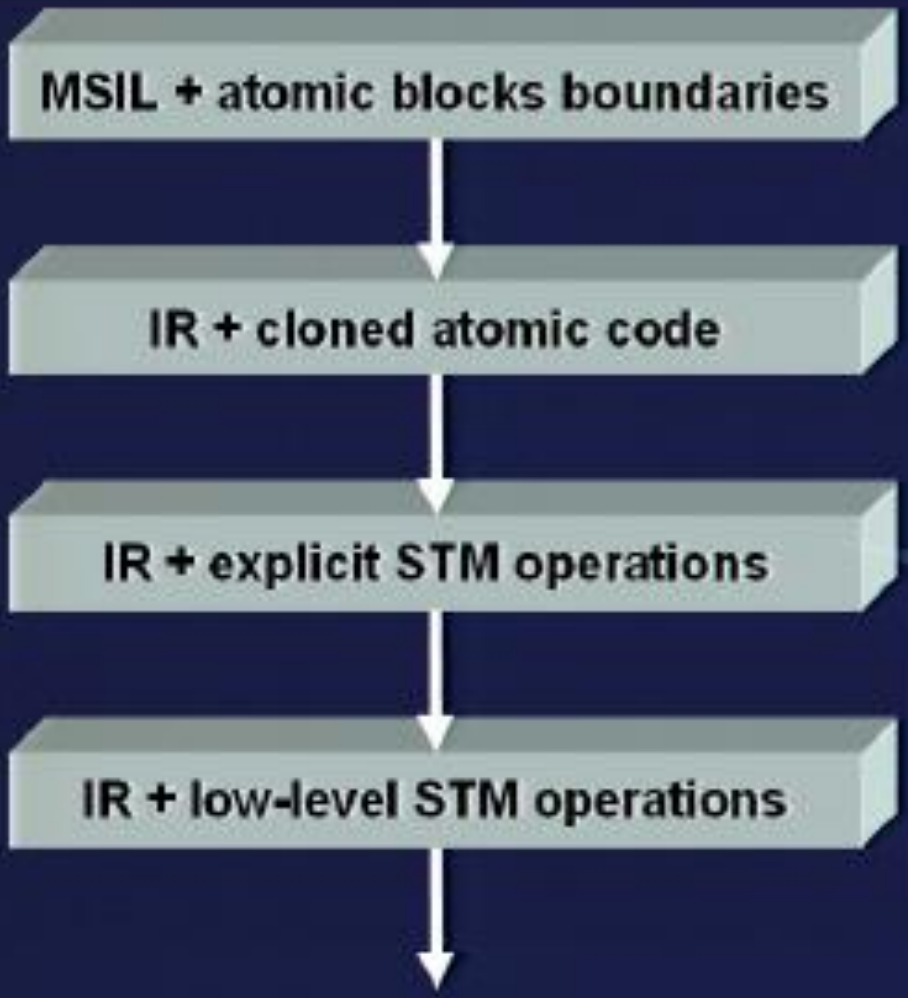

```
try {
  …
  if (node.Right != this.sentinelNode)
  …
} catch (AtomicFakeException) {
}
```

Compilation




```
l_1 = ObjectField_enlist_read<Right>(t_0)
t_1 = ObjectField_enlist_read<sentinelNode>(
t_2 = Neq<bool>(l_1, t_1)
```

Compilation




```
EnlistObjForRead(t_0)
l_1 = ObjectField<Right>(t_0)
EnlistObjForRead(l_0)
t_1 = ObjectField<sentinelNode>(l_0)
t_2 = Neq(l_0, t_1)
```

Three-pronged attack:

- Reduce the number of Enlist* and Log* operations that are executed
  - Remove unneeded operations
  - Use code motion to reduce execution frequency or to expose further optimization opportunities
- Amortize work between Enlist* and Log* operations
  - E.g. don't combine checks for space in the log
- Make what remains as fast as possible
  - E.g. expose fast paths for inlining from the STM runtime into the application

Some examples (xlisp garbage collector)

```
/* follow the left sublist if there is one */
if (livecar(xl_this)) {
  xl_this.n_flags |= (byte)LEFT;
  tmp = prev;
  prev = xl_this;
  xl_this = prev.p;
  prev.p = tmp;
}
```

Enlist 'prev' for udpate here to avoid an inevitable upgrade

Some examples (othello)

```
public int PlayerPos (int xm, int ym, int opponent, b
  int     rotate;                    // 8 degrees of rotatio
  int     x, y;
  bool endTurn;
  bool plotPos;
  int     turnOver = 0;

  // inital checking !
  if (this.Board[ym,xm] != BInfo.Blank)
    return turnOver; // can't overwrite player
```

Calls to PlayerPos must enlist 'this' for read: do this in the caller

Overview

Introduction

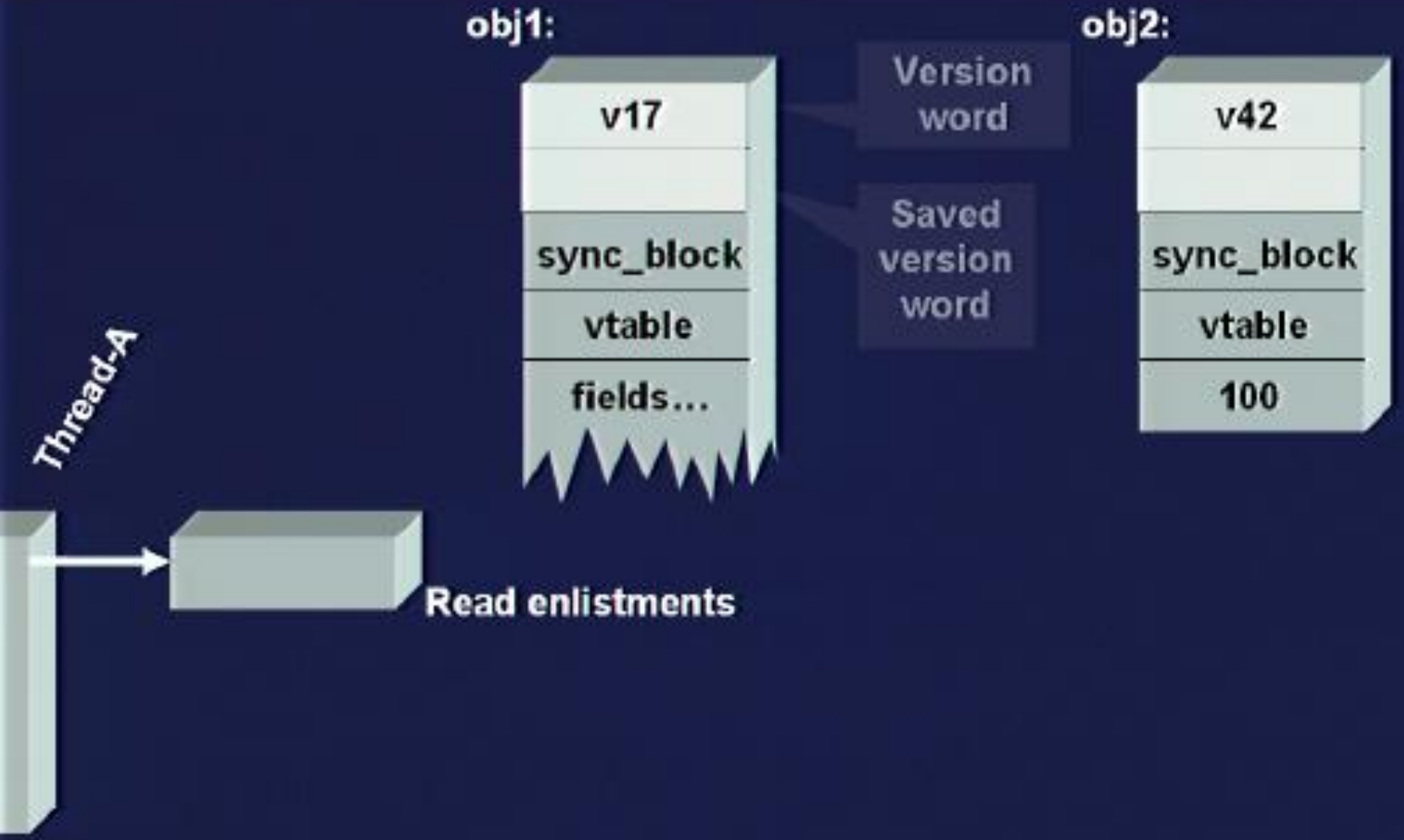
Optimizations

Runtime system

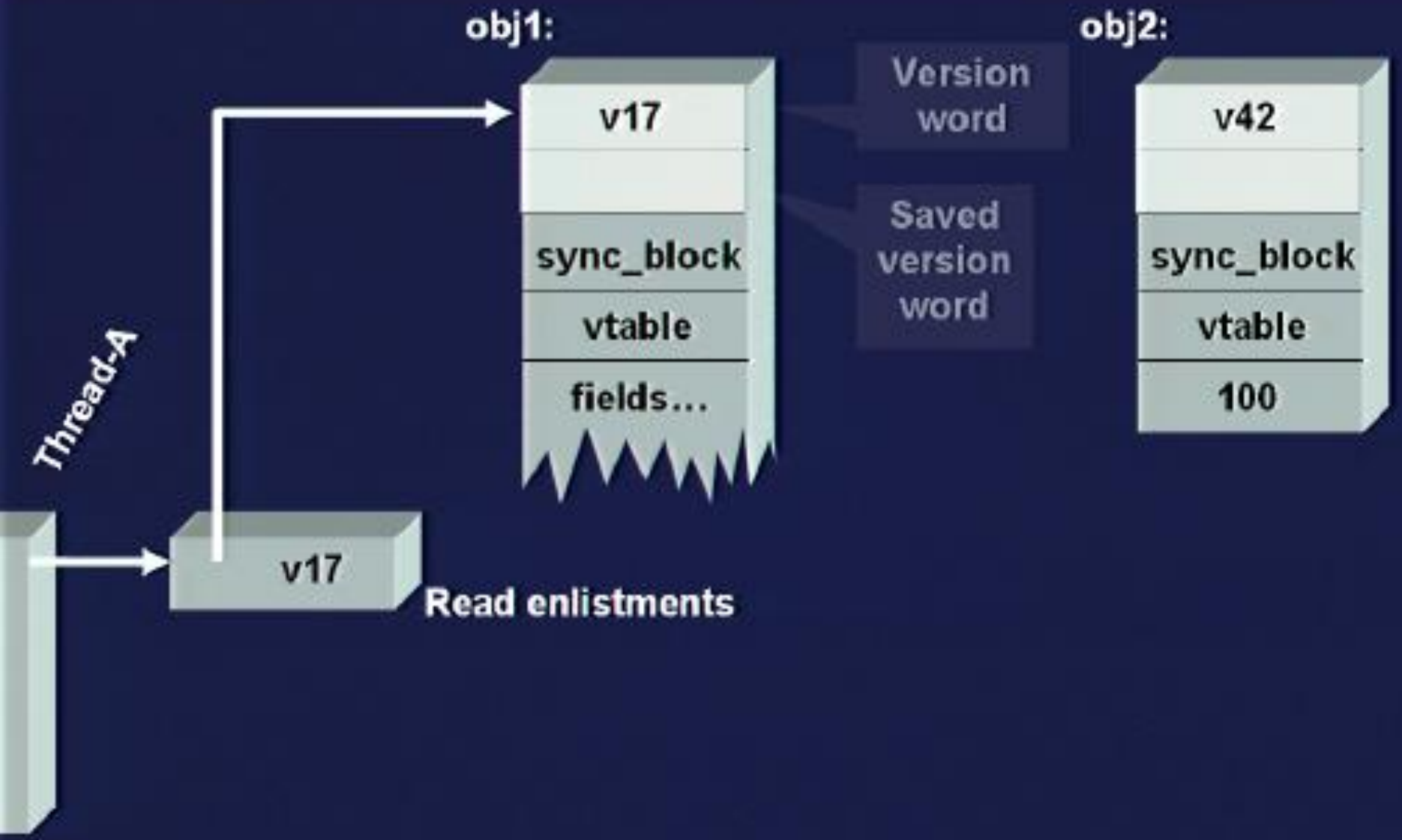
Initial results

Runtime system

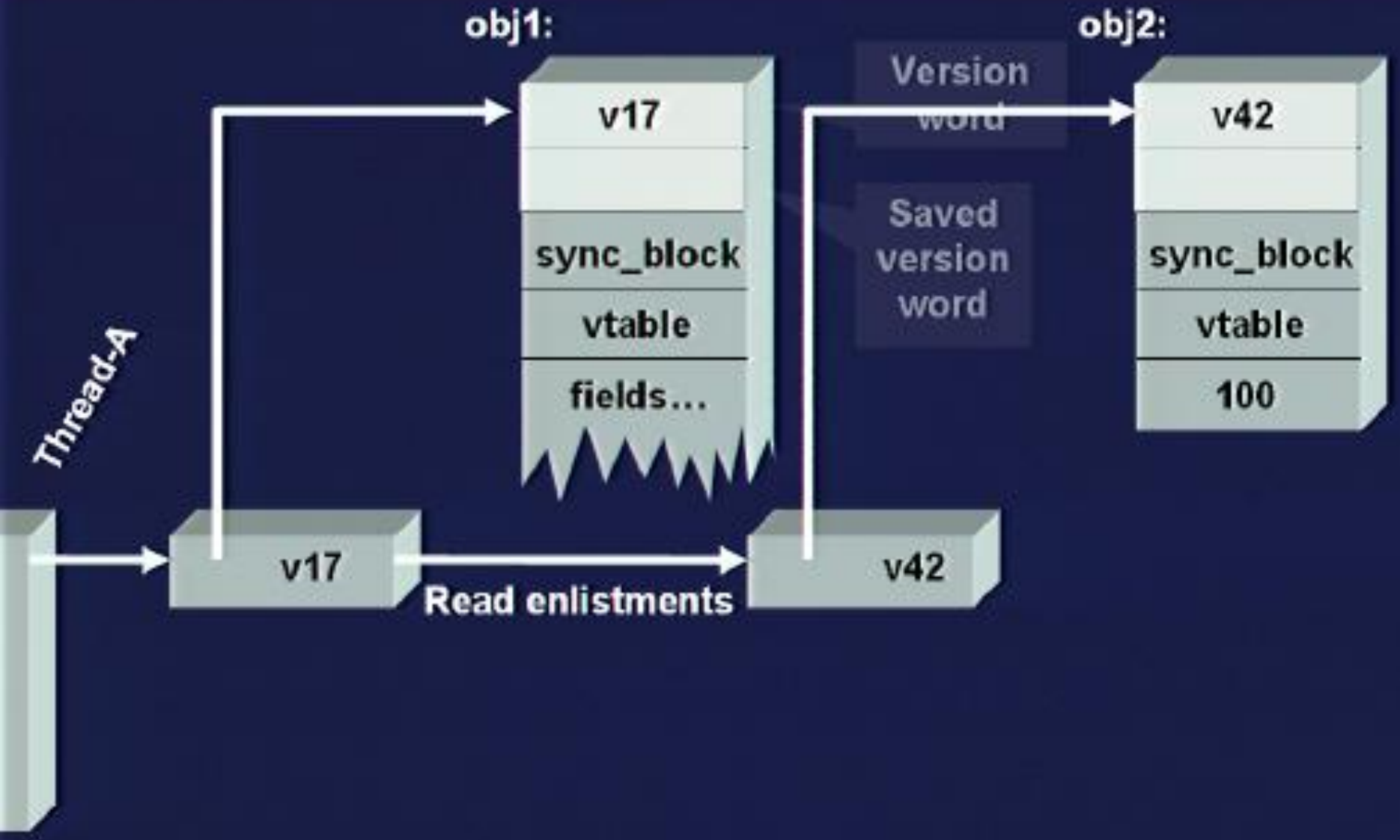
Runtime system



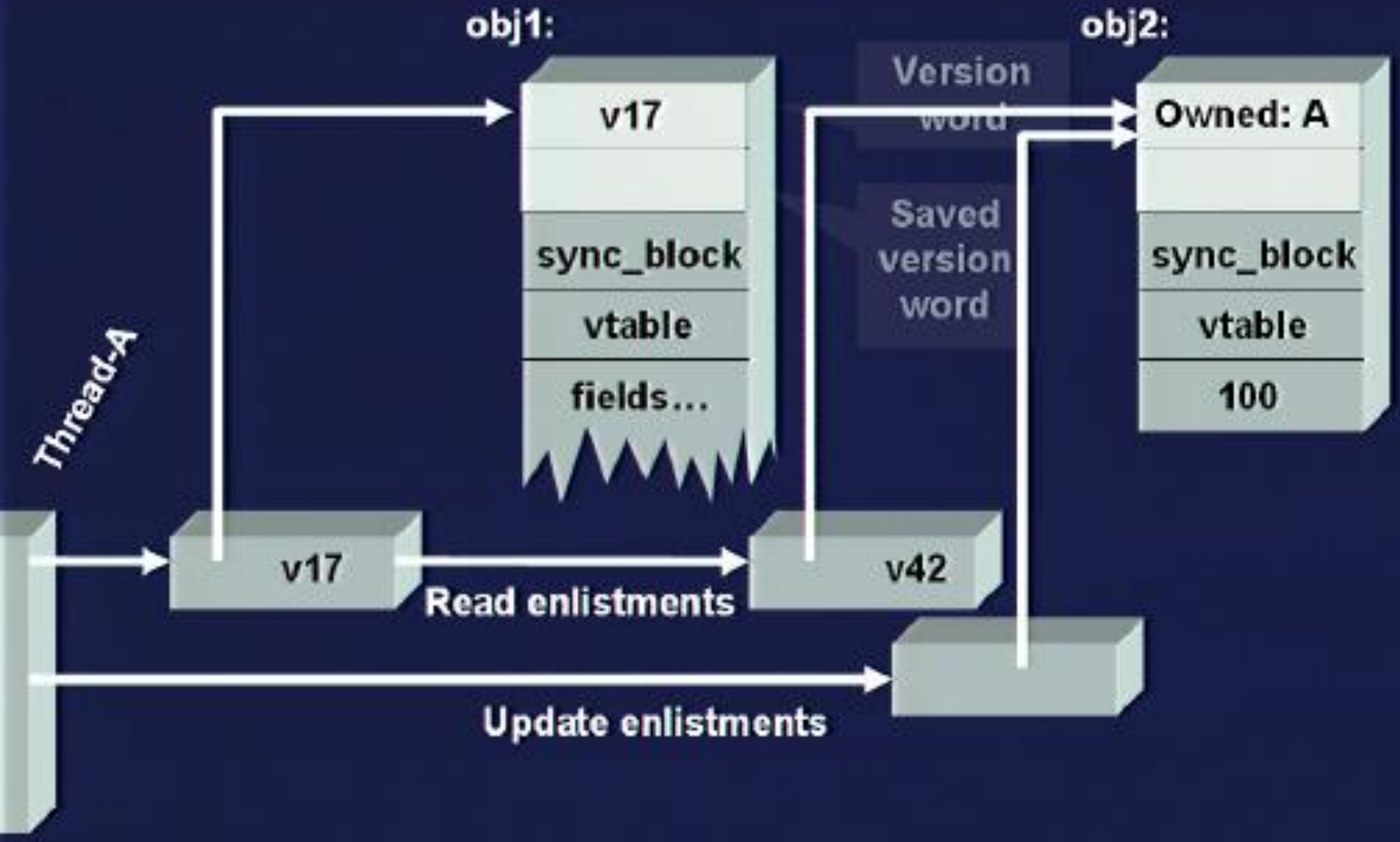
Runtime system



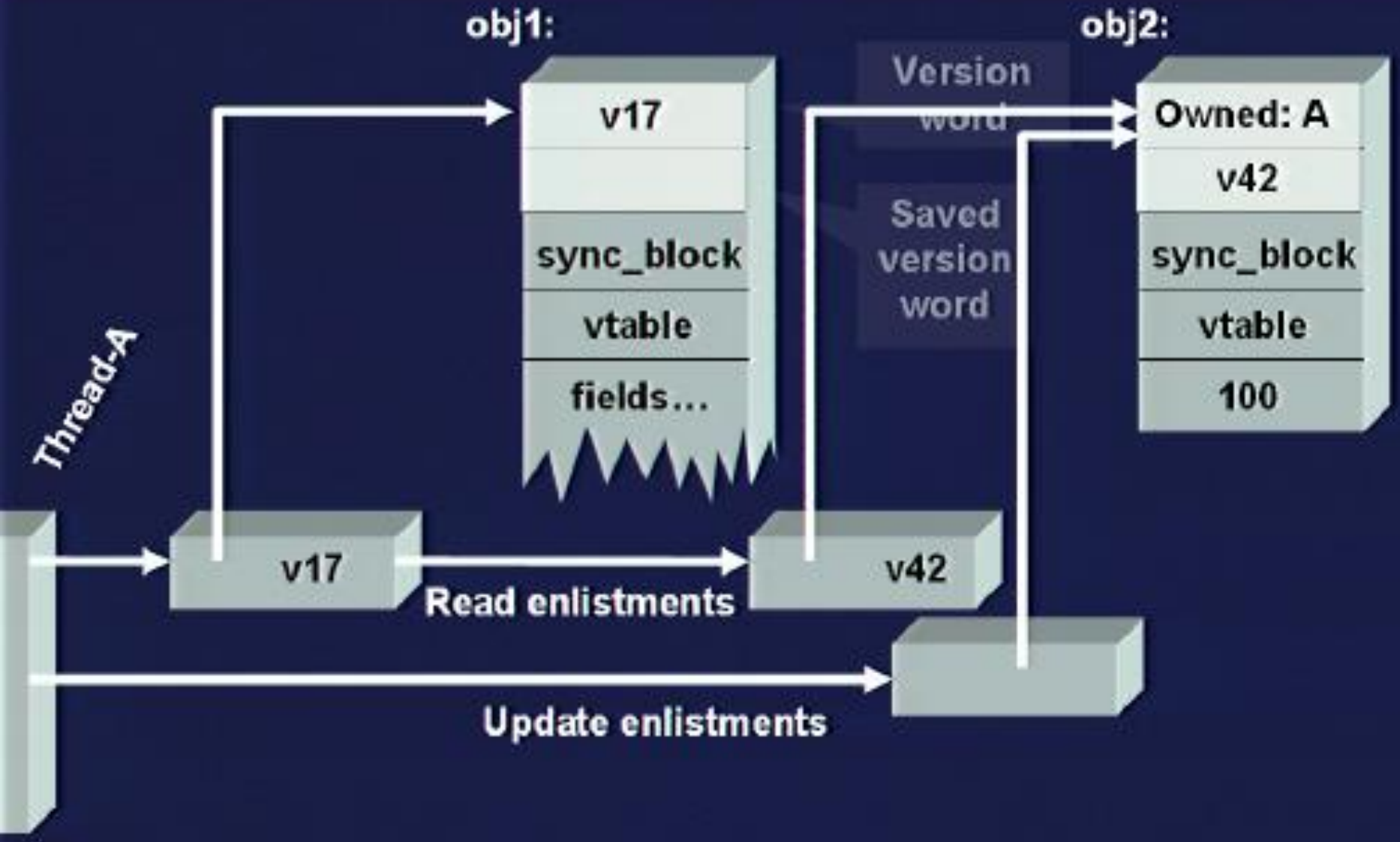
Runtime system



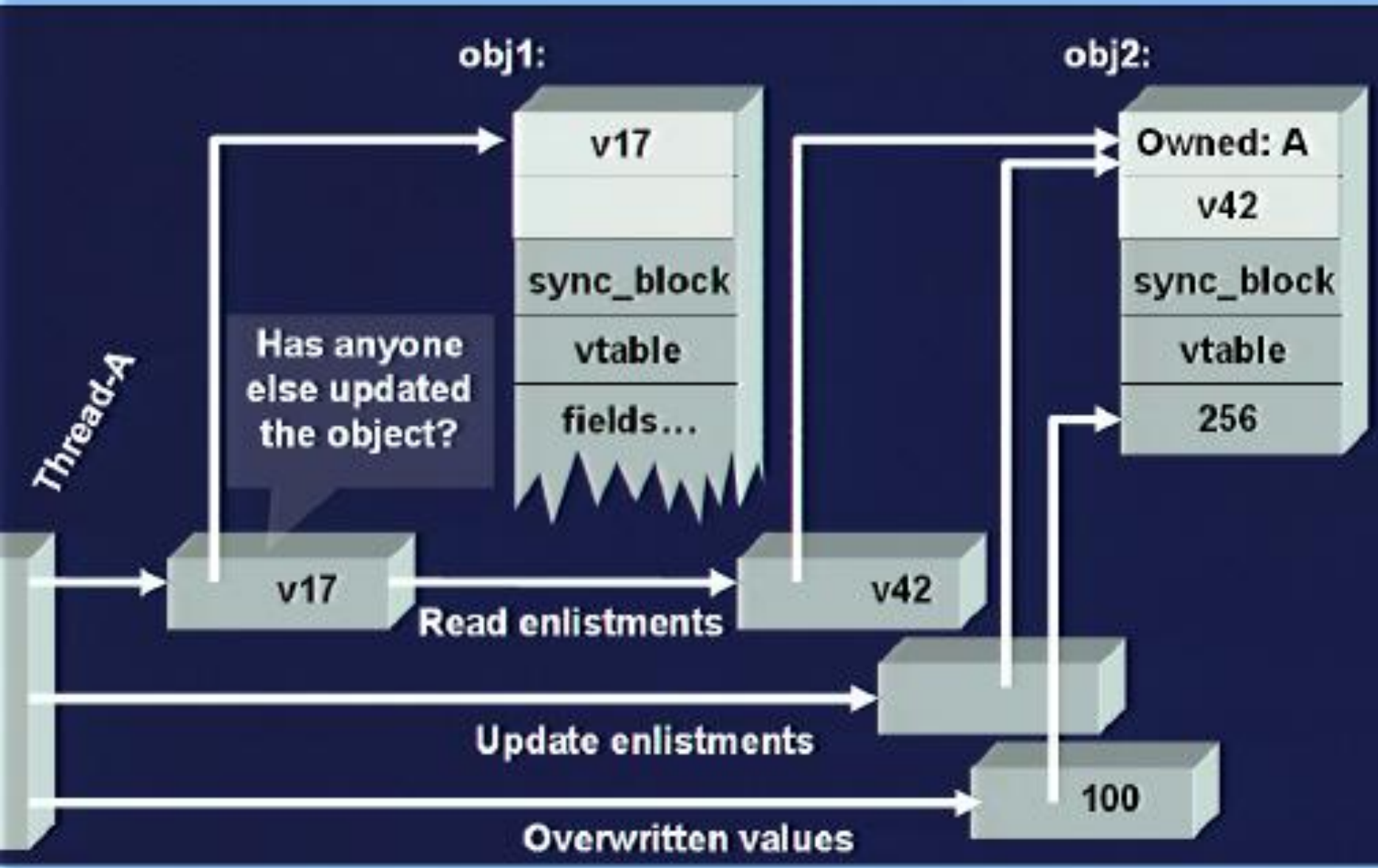
Runtime system



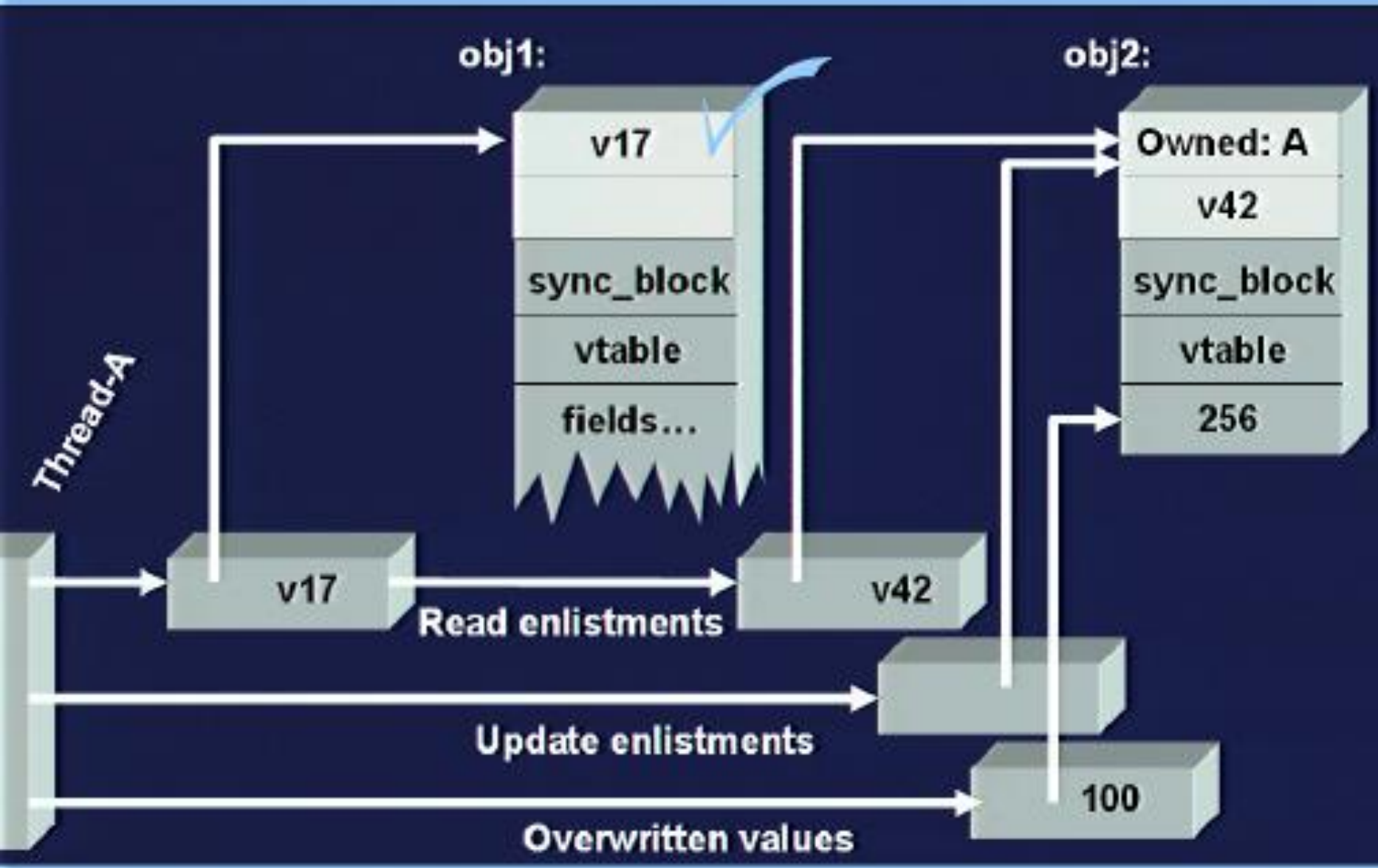
Runtime system



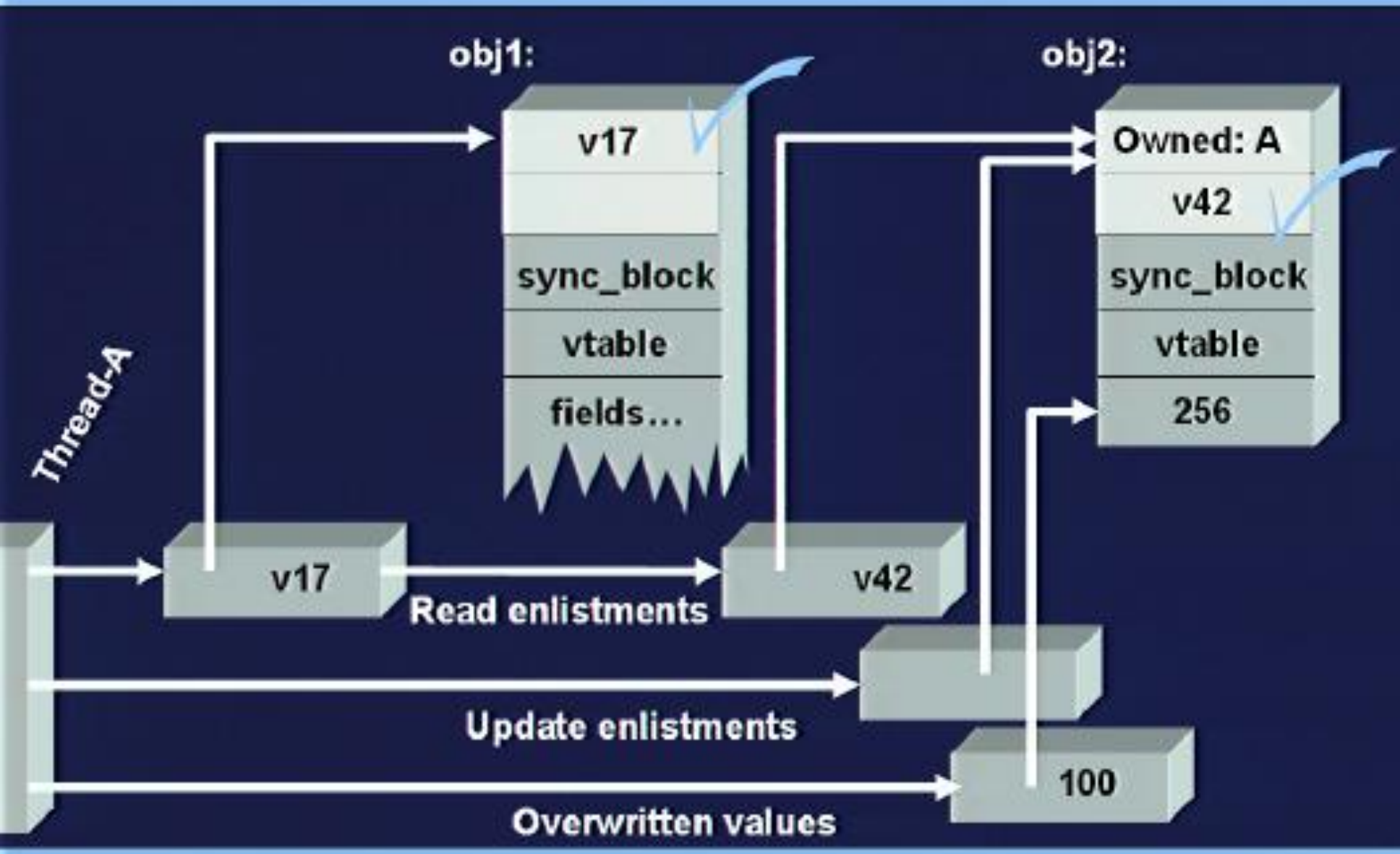
Commit


obj1:
v17
sync_block
vtable
fields…
obj2:
Owned: A
v42
sync_block
vtable
256
Thread-A
Has anyone else updated the object?
v17
Read enlistments
v42
Update enlistments
100
Overwritten values

Commit


obj1:
v17
sync_block
vtable
fields…
obj2:
Owned: A
v42
sync_block
vtable
256
Thread-A
v17
Read enlistments
v42
Update enlistments
100
Overwritten values

Commit


obj1:
v17
sync_block
vtable
fields…
obj2:
Owned: A
v42
sync_block
vtable
256
Thread-A
v17
Read enlistments
v42
Update enlistments
100
Overwritten values

What about... version wrap-around

Overview

Introduction

Optimizations

Runtime system

Initial results

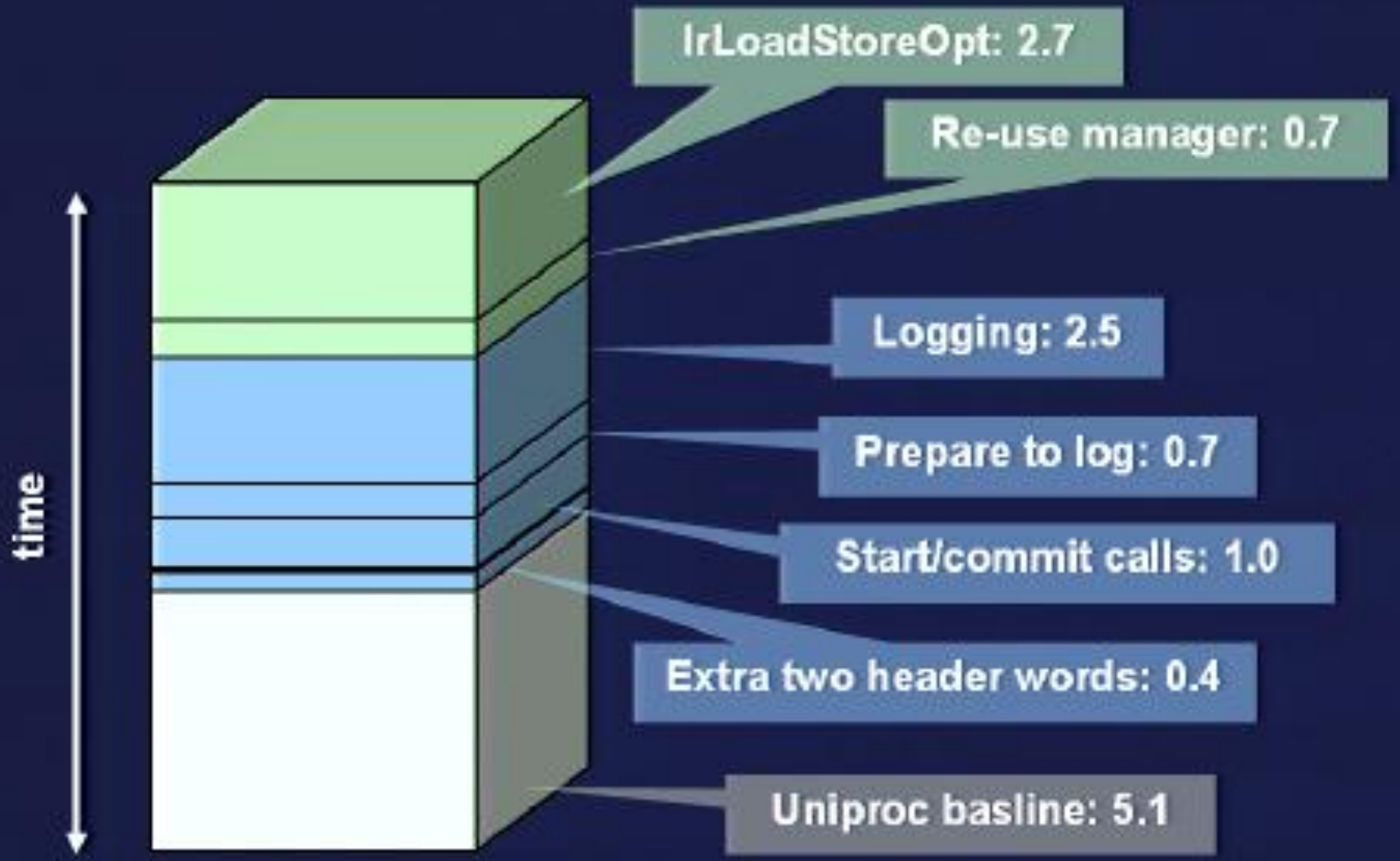
Evaluation

- Method: compare the slowdown when running uniproc code in an atomic block
- Why uniproc? STM design provides concurrency of non-conflicting transactions
- Two sets of tests:
  - Small concurrent programs: red black trees, skip lists. Aim for fast performance.
  - Large programs: SPEC-test derivatives. Look at how the slow-down scales as the size of the block gets larger – e.g. duplicate logging, GC work.

Implemented optimizations

- Retain TryAllManager object between Enlist* and Log* operations
- Treat Enlist* and Log* as expressions, use CSE
- Remove Enlist*ForRead if an Enlist*ForUpdate is available for the same location
- Move Enlist*ForUpdate earlier to avoid upgrades
- Extend existing intra-method null-check elimination to avoid logging on newly allocated objects
- Some cases of loop hoisting
- Inform the compiler of the granularity at which conflicts occur on statics

Red-black trees: 1.9x slower



XLisp: 3.9x slower (au boyer browse)

- Uniproc: 1.55s
- Atomic with current opts: 6.16s
- Atomic, no opts (runtime dup-elim): 10.8s

	Opt	No-opt
EnlistObjForRead	72	146
EnlistAddrForRead	6	23
EnlistObjForUpdate	27	44
EnlistAddrForUpdate	3	9
Logged enlistments	0.3	
LogHeapVal	19	25
LogHeapRef	23	28
Logged updates	0.6	

Status and future work

- Ongoing work
  - Many optimizations still to implement
  - Many settings compiled in differently for short / long transactions
- Checked in to Bartok
  - Ported to Singularity runtime Real Soon Now
  - …hopefully leading to experience with applications and insight into the language design issues
- Current goal is a PLDI submission (Nov)